FUSE'ing Python for Development of Storage Efficient Filesystem

Vishal Kanaujia
vishalkanaujia@gmail.com
Chetan Giridhar
cjgiridhar@gmail.com

Abstract

Filesystem is a core component of a functional operating system. Traditional Filesystem development has been confined to the kernel space. A customized, purpose-built, and user-driven Filesystem development involves extensive knowledge of kernel internals, tools and processes. Alternatively, user-space Filesystems are preferred over the kernel space Filesystem, for ease of development, portability and developing prototypes Filesystems, particularly for intuitive abstraction of "non-file" objects.

This paper proposes usage of FUSE kernel module to develop a functional Filesystem in user-space, titled "seFS". Apart from offering convenience of user-space development, FUSE allows on-par features and functionality of a kernel space Filesystem. We demonstrate development of a Filesystem in Python on Ubuntu 11.04 system with *Python-Fuse* bindings.

seFS Filesystem abstracts a SQLite database to store files data and metadata. By developing a Filesystem with Python-FUSE, we quickly solved the problem of efficient data management with online de-duplication and data compression. We discuss the internals of FUSE, its operation and implementation in this paper.

1. Introduction

A Filesystem manages data on a computer system. Filesystem is organized as a hierarchical system with directories at the top, containing a set of files, and a file itself being a collection of data blocks. Filesystem essentially maps name to an object and object to file contents. For instance, a file 'abc.txt' is a file name that maps to a file which eventually contains file data.

Thus, Filesystem provides a way to organize, store, retrieve and maintain information with the services like open(), read(), write(), close() among many.

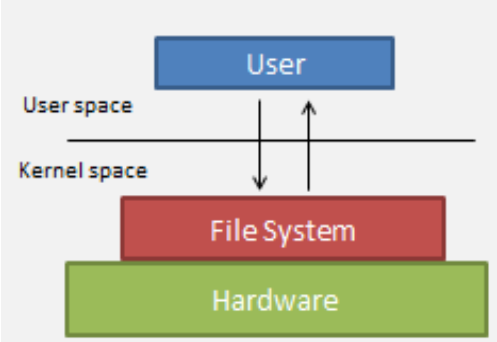


Figure 1: Block diagram representation of traditional Filesystem

Filesystems are essentially of two types:

- Real Filesystem
    - Store and manage data on disks or network
        - FAT32, NFS (Network Filesystem)
- Special Purpose Filesystem
    - Abstract a collection of entities as Filesystem
        - procfs

As shown in the Figure 1 above, it's important to note that a Filesystem implementation resides in the kernel space. This reduces user/kernel space switches and enables high performance of applications I/O.

So, how many Filesystems does a typical Linux box contain? If more than two, how are they managed? The answer is the Virtual Filesystem (VFS).

2. Filesystem in UNIX: Virtual Filesystem

In *NIX kernel, a Filesystem implementation is abstracted with a virtual Filesystem (VFS) [3]. VFS is an umbrella that acts as an interface to all available (mounted) Filesystems on a computer system. VFS itself has generic implementation of Filesystem operations. Its major task is to:

- Decouple Filesystem operation from the interface
- Manage Filesystem 'mount'
- Figure out the target Filesystem for an file I/O request and route the request
- Provide a consistent interface (POSIX) to user application for file I/O

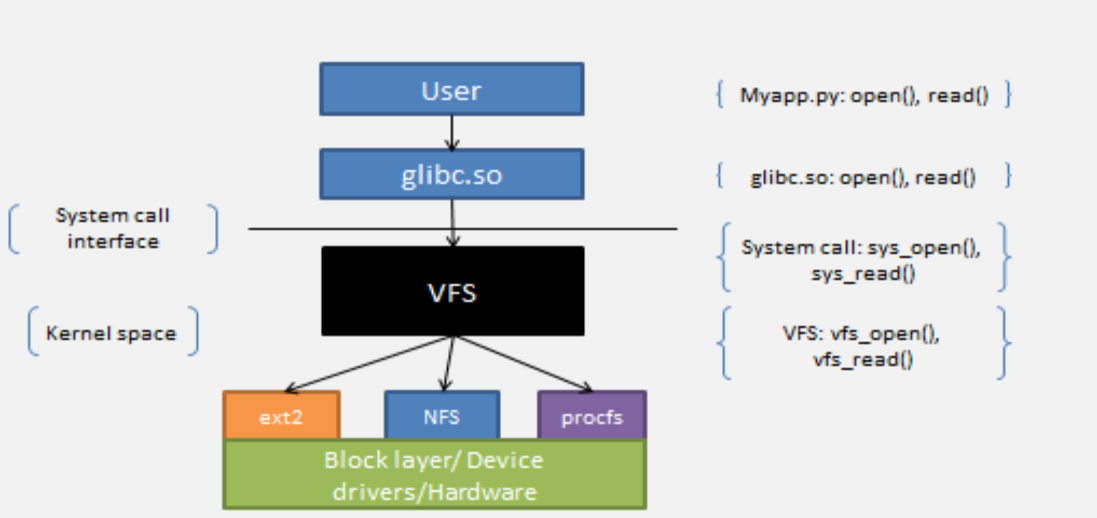


Figure 2: Flow diagram of Virtual Filesystem

Filesystems on *NIX system get registered with VFS with a mount point. What happens when a user application tries to access a file residing on (e.g. NFS partition) Filesystem? Let's understand it with Figure 2.

- Myapp.py calls open() with appropriate file path
- The call is directed to 'glibc.so' (user space library)
- 'glibc.so' makes a system level call (sys_open()) that is directed to the kernel space
- VFS handles this call and in turn triggers its own open() implementation which is vfs_open()
- Based on the mount point with which a Filesystem is registered with VFS, VFS identifies to which Filesystem the vfs_open() should be made.

3. User space Filesystem

Filesystems have traditionally been developed in Kernel space. Development in kernel space is complex in itself.

- It requires knowledge of kernel programming practices, kernel libraries and modules.
- Despite VFS, in-kernel Filesystems are difficult to port to another flavor of Linux
- Besides, development tends to be slow due to limited developer tools availability and frequent panics.

- Testing kernel code involves higher testing efforts.
- Dangers of kernel bloating and security threats are pertinent.

A user-space Filesystem can alleviate pains of development since it will become an ordinary user space application. This enables developer to use plethora of development tools and help shorten development time. Besides, user space Filesystem could offer portability, more control and flexibility to users. A user may choose to run a set of Filesystems, could share them with other users, without affecting the kernel.

4. FUSE

'Filesystem in **USE**rspace' is an open source, very stable, loadable kernel module for *NIX operating systems. It runs on Linux kernels 2.4.X and 2.6.X. This module facilitates user space Filesystem operations and exports a set of APIs to develop a fully functional, and non-privileged mountable user-space Filesystem. There are wrappers for FUSE APIs that are available in C, C++, Java, Python and many more languages to allow development for different applications.

FUSE based Filesystems securely co-exist with other mounted Filesystems. Unlike traditional Filesystems, FUSE allows a non-privileged mount of a Filesystem, and to share it with other users, securely. A FUSE based Filesystem has parity with kernel resident Filesystem in terms of functionality and features.

4.1 FUSE internals

FUSE is a loadable kernel module (fuse.ko) and acts as a Filesystem to the VFS. It registers itself with the VFS and opens a special device "/dev/fuse". FUSE module (fuse.ko) is an interface between the fuse device and the VFS. It receives file I/O requests from the VFS and writes those requests to "/dev/fuse" device. The use space library "lifuse.so" polls the device, and read "/dev/fuse" device.

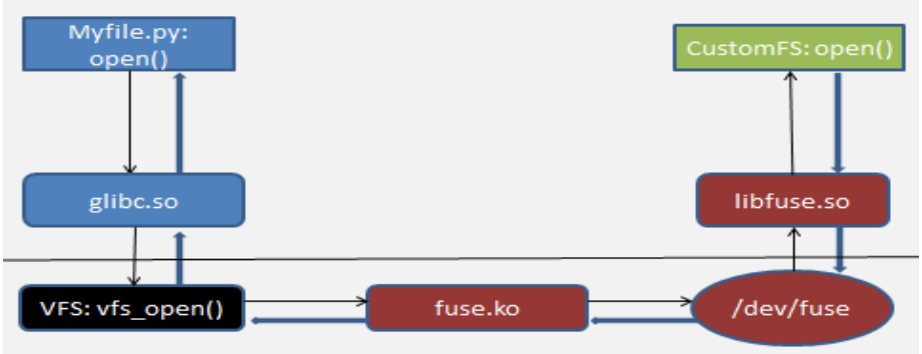


Figure 3: FUSE internals

The I/O request is propagated to user-space callbacks in Filesystem implementation. The response from the user space Filesystem is written back to "/dev/fuse", and "fuse.ko" sends the response to VFS that returns it to the application.

FUSE provides a helper utility *fusermount* to allow non-privileged users to mount the Filesystem.

4.2 FUSE Performance

FUSE based Filesystem has two user-kernel space switches for each I/O operation. There are two context-switches involved too for each I/O operation. Compared to user-kernel space switch, process context switches are expensive and impact performance of the Filesystem [9].

FUSE also introduces two additional memory copies in the kernel. When user space application writes data to a FUSE Filesystem, the data is copied to page cache and sent to libfuse via /dev/fuse and then from libfuse to page cache.

These factors may result FUSE based Filesystem unsuitable for high performance.

5. Developing storage efficient Filesystem (seFS)

A FUSE based Filesystem needs to implement FUSE API that appears similar to standard POSIX API. These APIs decide the behavior and functionality of the Filesystem. FUSE APIs are exported with FUSE bindings. Our FUSE based Filesystem is developed in Python with "FUSE-Python" binding. We chose to use the "Python-FUSE" binding as it is stable and actively developed. There are other bindings also including "FUSE-Python" and "fusepy".

A Filesystem operation is implemented by composing one or many FUSE-Python API functions. For example, a file read operation comprises getattr() followed by open() and read(). A FUSE based Filesystem needs to implement such APIs to accomplish file I/O operations. There is often one to one correspondence between "FUSE-Python" API and POSIX APIs; but a few functions like release () matches with close () of POSIX in functionality.

5.1 seFS Architecture

seFS Filesystem is designed to solve problem of better utilization of storage with data compression and de-duplication. The idea was to simplify data storage methods and focus more on data storage efficiency. seFS saves data and meta-data in a SQLite database and saves single instance of a file data. The instance is then stored in compressed form. The architecture of seFS is as following:

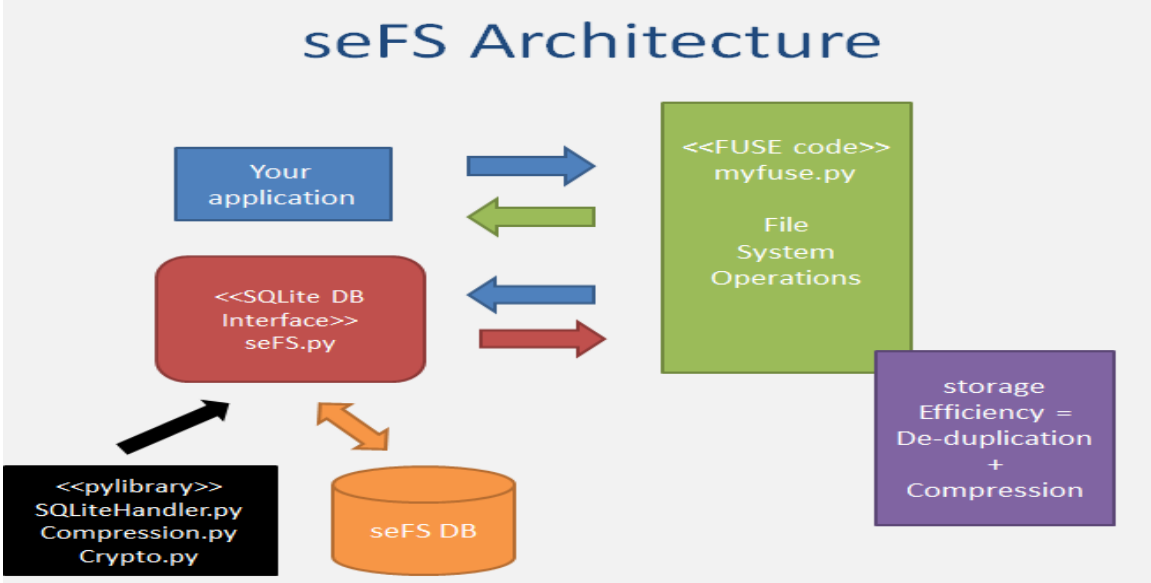


Figure 4: seFS Architecture

User application makes a call (e.g open()) to seFS (through VFS). seFS API (seFS open()) responds to this call based on its implementation. seFS APIs in turn interact with (SQLite) seFS DB for storage efficiency (data de-duplication and compression) with the help of SQLite DB interface seFS.py [4] and "pylibrary" [8] (home grown project that provides libraries for compression and hashing algorithms).

5.2 seFS Database

UNIX treats everything as a file. We leveraged FUSE to design a Filesystem backed by a database. By definition, a Filesystem is a layout of files and directories (In Linux systems, directories are also treated as files). seFS leverages the philosophy of 'treating anything as a file' in a way that SQLite Database itself is treated as a Filesystem. seFS database schema contains two tables:

- metadata
- data

```
CREATE TABLE metadata (
        "id" INTEGER,
      "abspath" TEXT,
     "length" INTEGER,
      "mtime" TEXT,
      "ctime" TEXT,
      "atime" TEXT,
     "inode" INTEGER);
```

```
CREATE TABLE data (
"id" INTEGER PRIMARY KEY
   AUTOINCREMENT,
      "sha" TEXT,
     "data" BLOB,
   "length" INTEGER,
 "compressed" BLOB);
```

Figure 5: seFS DB schema

In “metadata” table, all the metadata information of files (e.g. mtime, ctime, atime, inode, absolute file path) is stored. Each row in this table represents a file. “data” table has the actual file data in compressed format contents along with file length, SHA1 hash.

5.3 Storage Efficiency with seFS

seFS DB implements storage efficiency with:

- Data De-duplication
- Data Compression

Data de-duplication, also known as single-instance storage, is aimed at reducing storage requirements by eliminating data redundancy. For example, email system may contain multiple copies of various files that have been shared with multiple users. The data blocks of these files remain same and redundantly reside on the storage system.

Data de-duplication suggests storing single copy of the file and storing the reference for others. If an identical file is shared as n independent instances, the storage requirement of de-duplicated storage would reduce to 1/n$^{th.,}$ hence the storage efficiency.

Data de-duplication could be implemented on multiple levels of storage. It could be file based, block based or a mix of both. seFS attains de-duplication on a file level. In seFS, a file is created on FUSE based Filesystem (with command *touch*) by creating an entry in “metadata” table of seFS with relevant information (ctime, atime, mtime, inode, file absolute path information). If added file has data, a new row pertaining to the actual data, its length, its compression and SHA1 information is added in the “data” table of seFS. “data” and “metadata” tables reference each other with the primary key “id”.

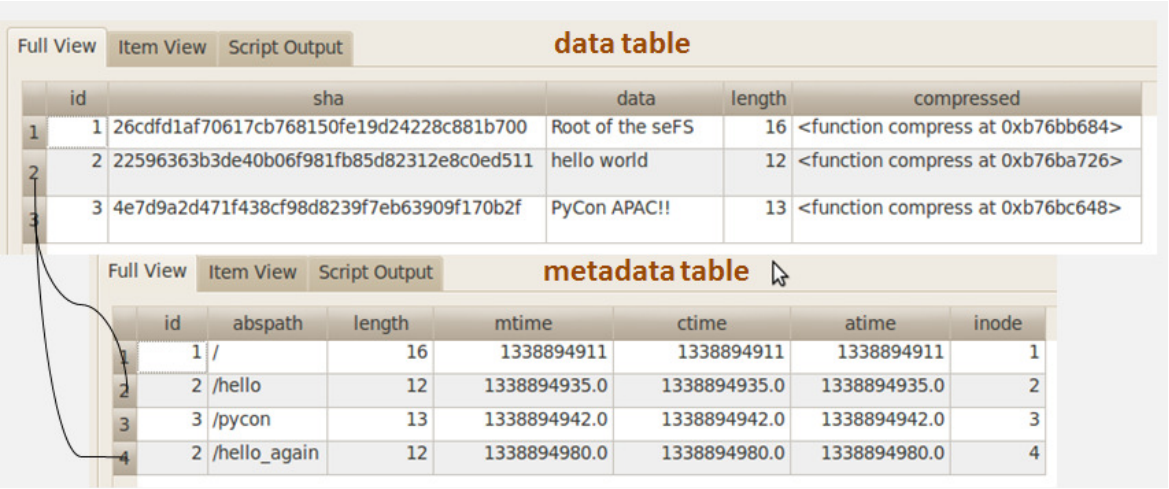


Figure 6: seFS Database

If we create a new file, a new entry gets created for this file in “metadata table”. If we add identical data to this file, a look up on seFS DB “data” table prevents creation of another entry in data table as the SHA information would match, and at the same time, the ‘id’ field of the “metadata” table for new file is updated as in Figure 6.

Compression reduces data size and it could be easily stored, retrieved and further processed. seFS uses a test based compression and stores compressed object in data table (compressed field). Data can be easily retrieved by decompressing the data with the help of compression object stored in data table.

5.4 seFS FUSE API

FUSE-Python binding exports a set of APIs to implement a Filesystem. A good understanding of these APIs is important to develop useful file I/O operations. An I/O operation is accomplished by composing one or more API functions; e.g. a POSIX read () operation is carried out by at minimum executing getattr (), open (), access (), and read () functions of your Filesystem.

FUSE provides flexibility to developers and requires implementation of only what you need. As an exception, all the APIs but getattr () are optional.

seFS Filesystem implements following operations:

- **__init__**()
  The Filesystem initialization involves creation of root (/). The root is created in the seFS database. It is necessary to have a root entry in a Filesystem. The following code snippet creates a root node.

```
class MyFS(fuse.Fuse):
    def __init__(self, *args, **kw):
        fuse.Fuse.__init__(self, *args, **kw)

        # Set some options required by the Python FUSE binding.
        self.flags = 0
        self.multithreaded = 0

        self.fd = 0
        self.sefs = seFS()
        ret = self.sefs.open('/')
        print "Created root with %s" %ret
        self.sefs.write('/', "Root of the seFS")
        t = int(time.time())
        mytime = (t, t, t)
        ret = self.sefs.utime('/', mytime)
        print ret
        self.sefs.setinode('/', 1)
        self.is_dirty = False
```

- **getattr**(self, path)
  This is the core and most used routine in any FUSE based Filesystem. It’s unavailability result in failure of the Filesystem. Its major work involves populating a

'file stat' object. The FUSE API provides stat object and for the given path, we need to fill in the details. seFS sets file times, mode, and length. We access this data from the seFS database.

```
def getattr(self, path):
    sefs = seFS()
    st = fuse.Stat()
    c  = fuse.FuseGetContext()
    print c
    print "getattr called path= %s", path
    if path == '/':
            st.st_nlink = 2
            st.st_mode = stat.S_IFDIR | 0755
    else:
            print "For a regular file %s" %path
            st.st_mode = stat.S_IFREG | 0777
            st.st_nlink = 1

    st.st_uid, st.st_gid = (c['uid'], c['gid'])

    ret = sefs.search(path)
    print "From database getattr ret=", ret
    if ret is True:
            tup = sefs.getutime(path)
            print tup
            st.st_mtime = int(tup[0].strip().split('.')[0])
            st.st_ctime = int(tup[1].strip().split('.')[0])
            st.st_atime = int(tup[2].strip().split('.')[0])

            st.st_ino     = int(sefs.getinode(path))
            print "inode = %d" %st.st_ino
            if sefs.getlength(path) is not None:
                    st.st_size = int(sefs.getlength(path))
            else:
                    st.st_size = 0
            return st
    else:
            return - errno.ENOENT
```

- **read**(self, path, size, offset)
  It handles read request from applications. The path is validated in the database and asked data is returned.

- **readdir**(self, path, offset)
  It implements the file listing functionality by querying the database for all the valid files. seFS is a flat Filesystem; there are no directories.

- **access**(self, path, flag)
  It is called by any routine that wants to ascertain existence of a given file. It runs a query in the database to search the given path.

- **open**(self, path, flags)
  Opens a file with given path by searching the file entry in the database.

- **write**(self, path, data, offset)
  The file write is bypassed to database writes. The seFS module creates a metadata entry for the file, calculates the SHA of passed data and searches for SHA in its data

table. If SHA match is found, the metadata table is updated with corresponding row ID from DATA table.

```
def write(self, path, data, offset):
    print "In write path=%s" %path
    length = len(data)
    print "The data is %s len=%d offset=%d" %(str(data), length, offset)
    sefs = seFS()
    ret = sefs.write(path, data)
    return length
```

- **release**(self, path, flags)
  There is no close() call in FUSE. The close() is accomplished by release().

- **create**(self, path, flags=None, mode=None)
  It creates an entry in the seFS database with the given 'path'. First, we try to open() the file and if it do not exist, we create the entry in the database. The access, creation, modification time and length is also set for this file.

```
def create(self, path, flags=None, mode=None):
    print "trying to create %s", path
    print path
    print flags

    sefs = seFS()
    ret = self.open(path, flags)
    print ret

    if ret == -errno.ENOENT:
            # Create the file in database
            ret = sefs.open(path)
            print ret
            print "Creating the file %s" %path
            t = int(time.time())
            mytime = (t, t, t)
            ret = sefs.utime(path, mytime)
            print ret
            self.fd = len(sefs.ls())
            print "In create:fd = %d" %(self.fd)
            sefs.setinode(path, self.fd)
            print sefs.ls()
    else:
            print "The file %s exists!!" %path
    return 0
```

- **unlink**(self, path)
  This routine implements file deletion. It deletes the entry from the seFS database.

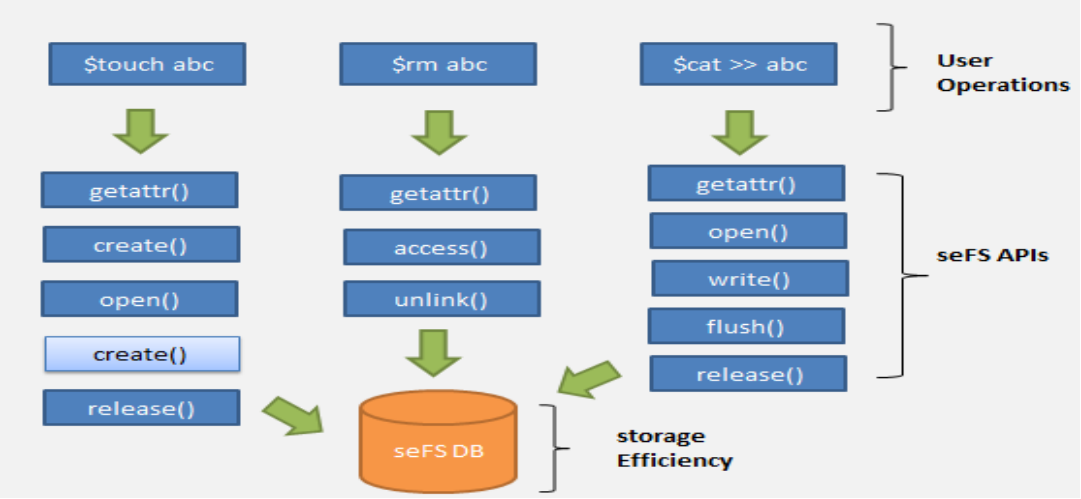


Figure 7: seFS API Flow

Figure 7 depicts seFS API flow when a user performs certain operations on the seFS Filesystem. For instance, "cat>>abc", appends data into the file abc. For this,

- seFS first getattr() to get the attributes on "abc" file,
- Opens the file by open()
- Writes the contents into the file with write()
- Flushes the buffer (flush()) and releases the file (release())

Similarly, when user deletes a file with 'rm' command, sequence of APIs followed is:

- getattr() is called to get file attributes
- file existence is checked with access()
- unlink() removes the file from seFS Filesystem

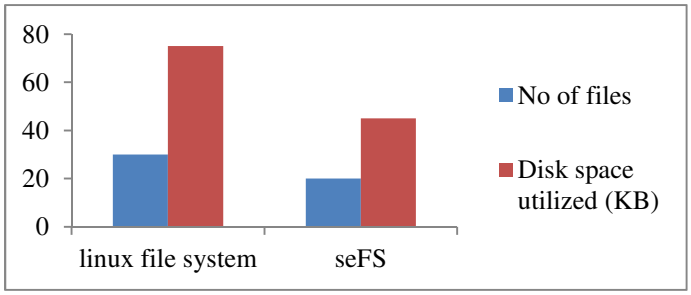
All these APIs internally interact with seFS DB interface and PyLibrary to achieve storage efficiency.

6. Analysis of Experiments

We leveraged seFS Filesystem in a software development system often referred as sandbox environment. During development, a lot of temporary or configuration files with identical data get generated on the Filesystem; because of which disk space usage was inefficient. The solutions to this problem could be:

- Delete files with shell scripts and cron
- Use seFS Filesystem to provide not only de-duplication but also additional storage efficiency with compression.

Reduction in disk space usage in a development system with seFS is shown below:



Parameters	Linux Filesystem	seFS
No of files	30	20
Disk space utilized (KB)	75	45

Figure 8: Table and chart manifesting reduction in disk space with seFS

The storage efficiency of seFS Filesystem is subjective to the dataset in use. Our analysis in this paper is to indicate that a set of files with similar content would get hashed to same value and thus we avoid creating duplicate copy when storing them.

The seFS file-system is essentially an interface to a SQLite database. The time efficiency of file operations in term of latencies for operations viz. create, move and delete are translated to SQL queries. Thus the time cost for these operations is database latencies for such operations.

7. Future implementation

The current seFS implementation is a flat Filesystem and we plan to evolve it to a complete, generic, and hierarchical structure with directories and sub-directories. We also need to evaluate the efficiency of data de-duplication and compression for various datasets. A more optimized and granular de-duplication at block level is also in our plans. Besides, we plan to improve Filesystem performance with caching, and profiling.

8. Summary

Traditional Filesystem development in kernel space demands steep learning curve is vulnerable to security holes and difficult to manage. FUSE ameliorates these problems and enables us to rapidly prototype a concept; develop a functional production level Filesystem with acceptable performance and on par features of kernel resident Filesystem.

Python bindings of FUSE are intuitive, robust and feature rich. Quick development with Python-FUSE is a logical extension to exploit the power of FUSE. It is a non-traditional way to develop a Filesystem with ease and flexibility of application domain programming.

FUSE is well suited for a prototype and feature rich Filesystem development but it might not favor a very high performance Filesystem as discussed in the paper earlier.

With FUSE Filesystem development (e.g. seFS) can focus more on designing the core features (de-duplication and compression) rather than fundamentally essential functionalities of a typical Filesystem. FUSE gives creativity of a Filesystem developer a new definition.

9. seFS project

seFS project is hosted on github. It can be accessed and forked from this location https://github.com/cjgiridhar/seFS. seFS is licensed under GNU General Public License v3.0

10. References

[1] Filesystem in User space, http://fuse.sourceforge.net/

[2] FUSE: https://en.wikipedia.org/wiki/Filesystem_in_Userspace

[3] VFS: http://tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html

[4] seFS: https://github.com/cjgiridhar/seFS

[5] Techtarget.com, articles on data de-duplication and compression

[6] Github.com, dedupfs - a project on de-duplicated Filesystem

[7] FUSE bindings http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FusePython

[8] Pylibrary, Custom libraries for Python developers, http://pylibrary.sourceforge.net/

[9] Performance and Extension of User Space File Systems: Aditya Rajgarhia, Ashish Gehani